VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Gaussian Blurring Mean Shift segmentace obrazu pomocí technologie NVIDIA CUDA

# Gaussian Blurring Mean Shift Image Segmentation Method Using NVIDIA CUDA Technology

2012 Vojtěch Cima

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání bakalářské práce

Student: **Vojtěch Cima**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Gaussian Blurring Mean Shift segmentace obrazu pomocí technologie NVIDIA CUDA
Gaussian Blurring Mean Shift Image Segmentation Method Using NVIDIA CUDA Technology

Zásady pro vypracování:

Student se musí seznámit s vývojem a charakterem technologie NVIDIA CUDA, strukturou procesorů a paměti. Dále je nutno nastudovat a naimplementovat segmentaci obrazu pomocí metody Gaussian Blurring Mean-Shift. Implementace musí být provedena jak v prostředí C++, tak CUDA. Práce bude obsahovat srovnání výsledků rychlosti obou kódů, náročnosti implementace.

Seznam doporučené odborné literatury:

http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf
http://drdobbs.com/cpp/207200659
Carreira-Perpi~nan, M.: Fast nonparametric clustering with Gaussian blurring meanshift. In: Proceedings of the 23rd international conference on Machine learning. pp.
153{160. ICML '06, ACM, New York, NY, USA (2006)
Comaniciu, D., & Meer, P. (2002). Mean shift: A robust approach toward feature space analysis. IEEE Trans. PAMI, 24, 603–619.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Milan Šurkala**

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 16. dubna 2012 Vojtěch Cima .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 16. dubna 2012 Vojtěch Cima .....

Rád bych touto cestou poděkoval Ing. Milanovi Šurkalovi za odborné a trpělivé vedení mé bakalářské práce.

## Abstrakt

Segmentace obrazu patří k základním pilířům počítačového vidění a zpracování obrazu. Clusterovací metoda Mean-Shift patří k těm výpočetně náročným, poskytující prostor pro návrh paralelního zpracování. Cílem této práce je seznámit se se současnými segmentačními metodami a technologií NVIDIA CUDA. Praktická část práce je zaměřena na návrh a efektivní implementaci algoritmu Gaussian Blurring Mean-Shift pro zpracování standardním procesorem i grafickým adaptérem za použití technologie CUDA. Součástí práce je porovnání jednotlivých implementací z pohledu jak dosaženého výkonu, tak náročnosti jejich vývoje.

**Klíčová slova:** CUDA, Mean-shift, Segmentace Obrazu, Optimalizace, Open MP

## Abstract

Image segmentation is one of the basic pillars of computer vision and image processing. Mean-Shift clustering method is the computationally challenging one, providing space for parallel algorithm design. The goal of this bachelor thesis is to become familiar with the state of the art segmentation methods and NVIDIA CUDA technology. The practical part of this thesis is focused on design and efective implemetation of Gaussian Blurring Mean-Shift algorithm for processing both the CPU and GPU using CUDA technology. Thesis contains a comparison of the various implementations both in terms of achieved performance and difficulty of their development.

**Keywords:** CUDA, Mean-shift, Image Segmentation, Optimization, Open MP

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| CPU | – | Central Processing Unit |
| CUDA | – | Compute Unified Device Architecture |
| DRAM | – | Dynamic Random Access Memory |
| GPGPU | – | General-Purpose computing on Graphic Processing Unit |
| GPU | – | Graphic Processing Unit |
| GBMS | – | Gaussian Blurring Mean-Shift |
| HT | – | Hyper-threading |
| RAM | – | Random Access Memory |
| TDR | – | Timeout Detection and Recovery |

# Obsah

# Seznam tabulek

# Seznam obrázků

# Seznam výpisů zdrojového kódu

# 1 Úvod

Segmentace jako taková je velmi obecnou úlohou, kdy ze vstupní množiny dat získáváme nějakým způsobem rozčleněnou množinu dat výstupních. Současné digitální obrazy jsou reprezentovány milióny diskrétních pixelů, které jsou jednotlivě nicneříkající o objektech vyskytujících se na nich. Segmentace obrazu si klade za cíl předzpracovat obraz tak, aby bylo jednodušší ho strojově zpracovat. Úlohy tohoto typu jsou snadnými pro lidi, či jiné živé bytosti se smyslovým vnímáním, ale i u nich může docházet k určitým individuálním odchylkám ve vnímání jednotlivých celků. Strojové zpracování obrazu a jeho segmentace je tedy výpočetní výzvou, jejíž výsledek je silně závislý na správné parametrizaci. Použití popsaných clusterovacích metod se uplatňuje mimo zpracování obrazu i na mnohem obecnější úlohy segmentace množin dat ve statistice nebo v oblasti dolování z dat. Výsledkem jsou segmenty sdružující prvky s podobnými vlastnostmi. Dále se již nemusíme zabývat každým prvkem množiny, stačí zpracovat segmentované oblasti a dosažené výsledky aplikovat na jejich jednotlivé prvky.

Fyzikální limity způsobují, že nelze zvyšovat výkon současných procesorů pouze zvyšováním jejich taktu. Velká spousta nově vznikajicích algoritmů zakládá svůj výkon na paralelním zpracování. Tomu nahrává i poměrně nízká cena zařízení umožňujících výpočty tohoto typu.

Cílem této práce je seznámit se s přehledem současně používaných metod segmentace obrazu se zaměřením na clusterovací metodu Blurring Mean-Shift s Gaussovým kernelem. Dále se pak věnuji popisu technologie NVIDIA CUDA a jejímu uplatnění při paralelních výpočtech. Implementační část této práce se zabývá efektivní implementací GBMS algoritmu určeného ke zpracování jak na procesoru CPU, tak jeho přepracováním pro výpočet pomocí GPU. Dále jsou porovnány výkony jednotlivých technologií a příslušných výpočetních jader.

Segmentace obrazu se v současné době široce uplatňuje v oblasti zpracování lékářských a vědeckých snímků, počítačového vidění, automatizace výroby v průmyslu, nebo rozpoznávání obličejů. Tato disciplína počítačové grafiky se stala nedílnou součástí současného života protkaného technologiemi. Využití v detekci a rozpoznávání obličejů je zvlášť kontroverzní. Napomáhá ke zvýšení bezpečnosti, ale ruku v ruce s tímto faktem dochází ke snižování obecného soukromí.

# 2 Segmentace obrazu

Cílem úlohy segmentace obrazu je zjednodušení vstupního obrazu tak, aby bylo snažší a rychlejší ho strojově zpracovat. Segmentací obrazu v oblasti počítačové grafiky rozumíme proces rozdělení obrazu na několik segmentů (množin pixelů) se společnými vlastnostmi a určitým významem. Sledovanými vlastnostmi pixelů bývají hodnoty jasu a jejich vzájemné vzdálenosti. Segmentace je prováděna skupinou metod (metody segmentace obrazu), které se liší výsledky, rychlostí a především stylem výpočtu. Je nutné zvolit vhodnou metodu segmentace obrazu pro požadované parametry výsledného obrazu. Hlavní myšlenkou je zjednodušit obraz tak, aby z něj byly patrné jednotlivé celky, které člověk vnímá automaticky a zároveň zanechat původní smysl obrazu.

Mezi základní typy metod pro segmentaci obrazů patří algoritmy prahování, clusterovací metody a metody založené na teorii grafů. Hlavním cílem této práce je seznámení se s algoritmy z rodiny clusterovacích algoritmů, zvláště pak s algoritmy Mean-Shift a Blurring Mean-Shift. Pro porovnání uvádím i stručný přehled a popis ostátních základních metod segmentace obrazu.

## 2.1 Prahování

Prahování je jednou ze základních metod zpracování obrazu. Obraz ve stupních šedi převedeme na binární dle zvoleného prahu (tresholdu). u barevných obrazů volíme treshold pro každou složku samostatně. Hodnoty jasu všech pixelů obrazu se porovnávají se zvoleným prahem. Následně se pixel přiřadí do jedné ze dvou skupin a tím například oddělí pozadí obrazu od popředí. Tato metoda má lineární složitost v závislosti na počtu pixelů. Je rychlá a často používaná. Problémem však bývá zvolení správné hodnoty prahu.

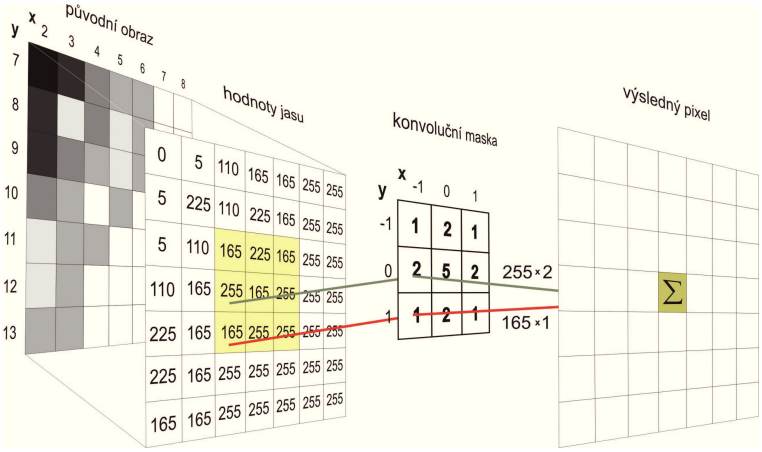
## 2.2 Metody detekce hran

Hrany v obraze lze definovat jako velké změny jasové funkce. Jako hrany tedy označujeme místa s velkou hodnotou derivace této funkce. Pro zjednodušení se však často používá metoda konvoluce s vhodným jádrem, která tuto derivaci aproximuje.

Pro představu uvažujeme diskrétní dvourozměrnou konvoluci, tedy použití konvoluční masky nad šedotónovým obrazem. Každý bod obrazu je postupně překryt středem konvoluční mřížky a výsledná hodnota nového pixelu je rovna součtu všech součinů překrývajících se pixelů mezi obrazem a maskou. Vhodně zvolená konvoluční maska pak zvýrazní oblasti s velkou změnou jasu, hrany.

## 2.3 Clusterovací metody

Spadají do kategorie algoritmů učení bez učitele. Není tedy dopředu znám tvar segmentů (clusterů) ani žádná jiná množina dat, ze které tyto algoritmy mohou vycházet. Slouží k shlukovaní prvků do skupin, kde prvky jedné skupiny jsou si nějakým způsobem nebo vlastností podobné a zároveň jsou odlišné od ostatních vzniklých skupin. Výpočet




Obrázek 1: Princip diskrétní dvourozměrné konvoluce.[6]

probíhá v jednotlivých krocích, iteracích. Mezi metody tohoto typu řadíme například algoritmy K-Means nebo Mean-Shift, kterým se tato práce zabývá podrobně.

### 2.3.1 K-Means

Clusterovací metoda K-Means přiřazuje každý z prvků vstupní množiny jednomu z k center (atraktoru) v následujících krocích:

1. Do vstupní množiny se náhodně umístí všechna centra.
2. Každý z prvků množiny vstupních dat se přiřadí k centru, které je nejblíže.
3. Centra se pohnou do středu prvků jim náležejících.
4. Body 2 a 3 se opakují dokud centra mění svou pozici.

Metoda K-Means s sebou přináší několik problémů. v první řadě je potřeba znát počet výsledných segmentů dopředu. Tato informace není ve všech segmentačních úlohách patrná na první pohled. Dalším problémem jsou situace, kdy algoritmus nevybere optimální řešení. Například 4 body tvořící obdelník o hranách $a, b$ a 2 centra ležící na středech protilehlých stran tohoto obdelníku. Výpočet je ukončen nehledě na rozdíly velikostí délek stran $a$ a $b$. s mnohorozměrnými daty roste dimenzionalita problému a jeho výpočetní náročnost.

# 3 Mean-Shift

## 3.1 Definice

Mean-Shift je bezparametrická, clusterovací metoda pro nalezení místa s největší hustotou vzorků. Není závislá na počtu ani tvaru segmentů. Probíhá iteračně, v každém kroku posouvá datový bod do váženého průměru bodů v okolí na základě funkce odhadu hustoty. Postupným posunem za středem vzorků se tento vektor posunu $m(x)$ zmenšuje až do chvíle, kdy je posun nulový nebo natolik malý, že se neprojeví na výsledku segmentace a výpočet ukončíme. Prvky vstupní množiny takto konvergují k bodům (atraktorům), které tvoří centra vzniklých shluků. [4]

Metoda Mean-Shift počítá gradient hustoty bodů v oblasti výpočetního okna každého prvku. Relevantnost bodů ve výpočetním okně je dána radikálně symetrickou funkcí (kernelem). v praxi tedy použitý kernel definuje váhu, s jakou budou prvky výpočetního okna započítány do výpočtu a jak ovlivní výsledný posun. Vektor posunutí $m(x)$ je zadán vztahem

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)} - x, \tag{1}$$

kde $N(x)$ je množina prvků okolí tj. prvky výpočetního okna a $K(x_i - x)$ je kernel, kde uvnitř výpočetního okna platí $K(x) \neq 0$.[1] Kernely bývají v praxi definované na omezeném intervalu $< -1, 1 >$. Je proto vhodné používat normované souřadnice bodů.

Pro jednoduchost pracujeme s obrazy ve stupních šedi, kde je každý pixel určen vektorem $[x, y, i]$. Složky $x, y$ reprezentují polohu pixelu ve 2D obrazu a prvek $i$ hodnotu jasu na příslušné pozici. Proto lze problém segmentace šedotónového obrazu řešit clusterovací metodou Mean-Shift pro 3-dimenzionální prostor. Blízké pixely v našem prostoru $[x, y, i]$ budou vytvářet shluky a je pravděpodobné, že tyto shluky reprezentují souvislé objekty z obrazu. v barevných RGB obrazech je Mean-Shift 5-dimenzionální problém, prostor se rozšíří o barevné informace. Vektor každého pixelu tedy bude $[x, y, r, g, b]$.

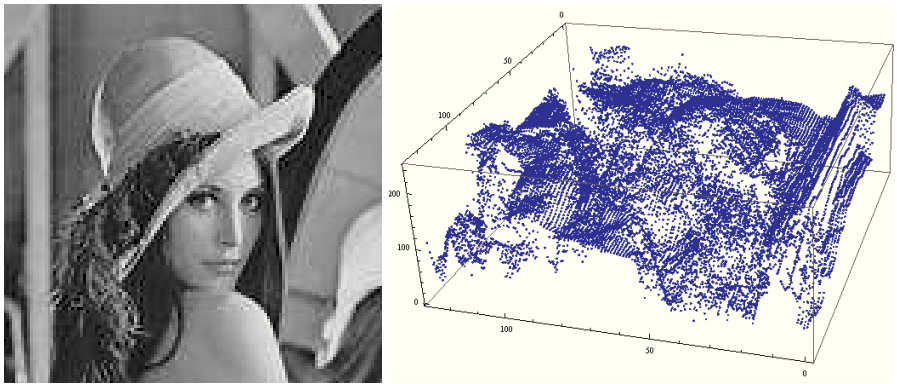
## 3.2 Blurring Mean-Shift

Blurring verze algoritmu Mean-Shift lehce upravuje myšlenku původního algoritmu. Na rozdíl od standardního Mean-Shift algoritmu, kde se po celou dobu výpočtu pracuje nad stejnou množinou, v každé iteraci pracujeme nad výslednou množinou iterace předcházející. k výsledné množině se dopracujeme v jednotlivých iteracích prováděných postupně nad celou množinou prvků. Vektor posunutí $m(x)$ u Blurring Mean-Shift algoritmu popisuje rovnice

$$m(x_{\tau+1}) = \frac{\sum_{x_i \in N(x_\tau)} K(x_i - x) x_i}{\sum_{x_i \in N(x_\tau)} K(x_i - x)} - x_\tau, \tag{2}$$

kde $N(x_\tau)$ je množina prvků okolí tj. prvky výpočetního okna v dané iteraci $\tau$ a $K(x_i - x)$ je kernel, kde uvnitř výpočetního okna platí $K(x) \neq 0$.[1] Stejně jako v případě klasického MS pracujeme s normovanými hodnotami souřadnic.

V každé iteraci dojde pouze k jednomu posunutí v rámci každého prvku celé množiny. Tato výsledná množina se stane vstupní a postup se opakuje.



(a) (b)

Obrázek 2: Ukázka obrazu a jeho znázornění v prostoru. 2a) Šedotónový obraz 128×128. 2b) Jeho odpovídající znázornění v 3D prostoru.

### 3.2.1 Kritérium zastavení výpočtu GBMS

Vzhledem k faktu, že v každé iteraci pracujeme nad změněnou vstupní množinou, nemůžeme výpočet zastavovat postupně pro jednotlivé prvky jako v případě původního Mean-Shift algoritmu.[3] Zastavení výpočtu pouze na základě dostatečně malého průměrného posunu prvků tedy u GBMS nelze aplikovat, protože k této situaci nemusí vždy dojít. Mnohem robustnější podmínkou zastavení výpočtu GBMS je porovnávat entropii histogramů pozic bodů mezi dvěma následujícími iteracemi. Průběh výpočtu GBMS se typicky skládá ze 2 fází. v první dochází k vytváření jednotlivých clusterů, tedy k námi potřebné segmentaci. v druhé fázi se však počet prvků náležejících jednomu atraktoru nemění a dochází pouze k postupnému pohybu atraktorů směrem k sobě. Rozdíl entropií histogramů mezi následujícími iteracemi v druhé fázi výpočtu GBMS je proto nulový a výpočet můžeme ukončit. Pozdní zastavení výpočtu GBMS tedy může vést až k situaci, kdy všechny body náleží jedinému atraktoru a tvoří tak jeden nicneříkající segment. Toto platí především pro kernely pokrývající celou datovou množinu. Podmínka zastavení výpočtu GBMS je tedy definována následovně:

$$(|H(e^{\tau+1}) - H(e^{\tau})| < 10^{-8})\ nebo\ (\frac{1}{N}\sum_{n=1}^{N} e_n^{\tau+1} < 10^{-3}), \tag{3}$$

kde $H(e)$ je entropie histogramu dat v iteraci $\tau$ zadaná vztahem

$$H(e) = -\sum_{i=1}^{N} f_i \log f_i. \tag{4}$$

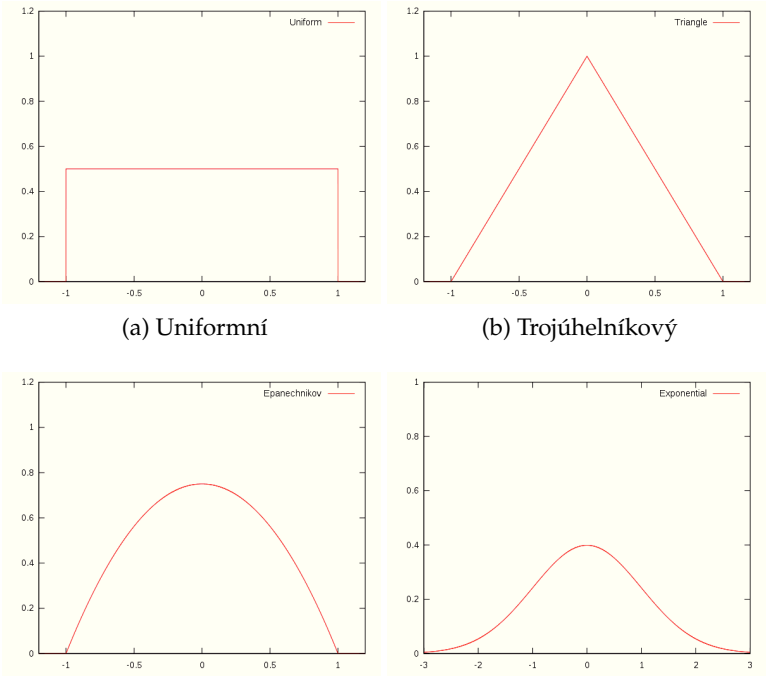
$f_i$ je relativní frekvence výskytu jednotlivých bodů, pro kterou platí $\sum_{i=1}^{N} f_i = 1$. Kontrola podmínky zastavení výpočtu má lineární složitost $O(n)$, která je v porovnání se složitostí výpočtu zanedbatelná.[1]

## 3.3 Kernel

Radikálně symetrická funkce kernelu $K(u)$ určuje váhu, s jakou budou prvky výpočetního okna započítány do výpočtu na základě jejich vzdálenosti v prostoru. Rozdílné typy použitých kernelů mohou přímo ovlivnit výslednou množinu i rychlost výsledku související se šířkou, jakou oblast kernel pokrývá. Uvádím několik často používaných typů kernelů. Jejich průběhy jsou znázorněny na obrázku 3.

| | |
|---|---|
| Uniformní kernel | $K(u) = \frac{1}{2} 1_{\{\|u\| \leq 1\}}$ |
| Trojúhelníkový kernel | $K(u) = (1 - \|u\|) 1_{\{\|u\| \leq 1\}}$ |
| Epanečnikův kernel | $K(u) = \frac{3}{4}(1 - u^2) 1_{\{\|u\| \leq 1\}}$ |
| Gaussův kernel | $K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}$ |

Tabulka 1: Typy používaných kernelů.




(a) Uniformní (b) Trojúhelníkový

(c) Epanečnikův (d) Gaussův

Obrázek 3: Průběhy kernelů. [5]

# 4 CUDA

## 4.1 Popis

Compute Unified Device Architecture je technologií společnosti NVIDIA často v literatuře uváděna pod zkratkou CUDA. Jedná se o rozhraní pro práci s grafickým adaptérem, rozšiřující stávající programovací jazyky C/C++, Fortran o možnost využití GPU k náročným výpočtům dosud probíhajících na CPU. Do uvedení technologie CUDA bylo programování pro GPU převážně záležitostí specifických úloh využívající grafiku, jakými jsou například CAD programy a 3D hry. Ukázalo se, že spoustu těchto úloh lze transformovat na obecnější výpočty a došlo ke vzniku GPGPU (General-purpose computing on graphics processing units).

Velký počet nezávislých výpočetních jednotek současných grafických adapterů poskytuje řádové zlepšení výkonu aplikace využítím masivního paralelismu, současného běhu stovek tisíc nezávislých vláken. Programování s využitím vláken využívají procesory CPU již řadu let. Nicméně výkony takovýchto výpočtů jsou omezené malým počtem výpočetních jader a široké škále funkcí klasického CPU, na rozdíl od GPU orientovaných na rychlé zpracování několika typů úloh. Vlákna tedy často využívala spíše pseudoparalelismu, program sice rozdělí do více větví, což se příznivě projeví na čitelnosti a logice kódu, ale zpracování bývá sekvenční. Zatímco CPU je především optimalizováno pro nízké latence jednoho vlákna se širokými možnostmi uplatnění, GPU je zaměřeno na co největší propustnost mnoha vláken.

Fakt, že se jedná o pouhé rozšíření stávajících programovacích jazyků, napomáhá k širšímu použití této technologie. Největším limitačním faktorem technologie CUDA je její použitelnost pouze na zařízeních společnosti NVIDIA. Současně jsou tedy vyvíjeny i jiné technologie pro použití paralelismu v aplikacích, mezi které patří například technologie OpenCL. OpenCL definuje standard abstraktního hardwarového zařízení, který splňuje většina současných klasických procesorů, grafických procesorů ale i signální a některé mobilní čipy. Zároveň je definováno softwarové rozhraní přístupu k těmto zařízením. OpenCL tedy má širší možnosti nasazení, roztroušeností použitelného HW však může docházet ke kompromisům ve výkonu aplikace. Konkurenční platformu pro GPGPU přináší i firma AMD (ATI) dříve známou pod názvem "ATI Stream Technology". v současné době nese tato technologie název "AMD App Acceleration"a funguje, podobně jako technologie CUDA, pouze na adaptérech mateřské firmy, kterou je v tomto případě AMD.

Je poměrně snadné napsat fungující program pro architekturu CUDA, ale je nutné mít hluboké znalosti o GPU, aby byl výsledný kód efektivní. Následující řádky tedy popíší CUDA architekturu jak z pohledu hardware, tak vývoje pro ni určeného software.

Základní vlastnosti CUDA technologie:

- Vysoký výkon a propustnost u paralelních algoritmů.
- Škálovatelný model paralelního programování.
- Možnost heterogenního sério-paralelního programování.

- Minimální rozšíření jazyka, přidávající mnoho výhod.
- Omezení pouze na GPU firmy NVIDIA.

## 4.2 Architektura CUDA

Hardwarová architekura většiny GPU kompatibilních s CUDA technologií využívá podobného modelu rozložení výpočetních jader, který zrcadlí požadavky na programátora. Větší výkon současných GPU od předchozích, taktéž kompatibilních karet, je docílen především zvyšováním počtu multiprocesorů, optimalizací práce s paměťmi a jejich hierarchií. Základní výpočetní jednotkou současných grafických adaptérů firmy NVIDIA je multiprocesor. Pro lepší názornost si popíšeme architekturu konkrétního čipu NVIDIA GF 100 s 16 multiprocesory.

Multiprocesor (NVIDIA GF 100):

- 32 výpočetních jader, procesorů.
- 64KB RAM rychlé sdílené paměti mezi jádry.
- Registry s přímým přístupem výpočetních jader dosahující propustnosti v řádech GB/s.

Hardwarovové architektuře se podmaňuje způsob práce s vlákny. Jednotlivá vlákna programu jsou spouštěna na výpočetních jádrech. z předchozího popisu fyzického rozložení výpočetních jader je žádoucí seskupovat vlákna tak, aby měla společný přístup k co možná nejrychlejší paměti, pokud se potřebují dělit o informace, kterými jsou například mezivýsledky výpočtu. Hierarchie pamětí je popsána níže v samostané kapitole.

Vlákna jsou v programu seskupována do bloků a každý blok je zpracován na právě jednom multiprocesoru. Vlákna v tomto bloku sdílejí paměť s velmi krátkou přístupovou dobou označovanou jako "sdílená paměť", "shared memory". Počet vláken v bloku je omezen na 1024. Tato vlákna mohou být mezi sebou synchronizována. Bloky vláken jsou uspořádány do gridu a navzájem mezi sebou mohou využít pouze globální paměť, která je řádově pomalejší. [8]

Funkce a procedury CUDA programu jsou rozšířeny o označení, jestli se budou vykonávat na CPU nebo GPU. Zavedena jsou 3 označení:

- *__host__*
  Jedná se o výchozí nastavení jazyka C/C++. Kód je volán z CPU a na něm taky vykonán.
- *__device__*
  Určuje funkce určené pro běh pouze na GPU. Takto označené segmenty lze volat jen z globálních částí kódů, kernelů. Na rozdíl od nich však mohou vracet hodnotu.
- *__global__*
  Slouží výhradně pro označení procedury kernelu. Takto označená procedura je

volána z CPU, tedy pouze z částí kódu označeného jako *__host__*. Volání globálních procedur je rozšířeno o konfiguraci počtu a rozmístění vláken. Kód je pak proveden na GPU nezávisle pro každé z vláken. v rámci kernelu lze volat pouze funkce a procedury označené jako *__device__*.

CPU i GPU mají samostatné paměťové prostory s vlastní hierarchií. CPU část kódu může spravovat globální část paměti speciálními funkcemi pro alokaci/dealokaci paměťového prostoru, kopírování dat z RAM do DRAM a naopak. Hierarchie pamětí v CUDA zařízení je popsaná v další kapitole.

Pro alokaci paměti na zařízení a její uvolňení poskytuje tato technologie sadu funkcí. Stejně tak je k dispozici funkce, sloužící ke kopírování dat mezi jednotlivými paměťovými prostory. Pro usnadnění parametrizace těchto funkcí je definováno několik výčtových typů, které reprezentují často používané vlastnosti.

Principiálně lze běh programu využívající CUDA popsat v několika základních krocích:

1. Spuštění aplikace.
2. Alokace a kopírování dat do globální paměti GPU.
3. Spuštění CUDA kernelu nad vstupními daty.
4. Kopírování výstupních dat zpět z GPU
5. Zpracování výsledků, ukončení aplikace.

## 4.3 CUDA kernel

Kernel je funkční kód programu pro technologii CUDA, k jehož zpracování dojde v každém paralelně spuštěném vláknu. Jedná se o funkce označené klíčovým označením *__global__* před definicí samotné procedury. Jednotlivá běžící vlákna lze identifikovat, nelze však určit pořadí jejich ukončení. Označení *__global__* před definicí určuje, že procedura je volaná na CPU, její vykonání je však předáno GPU. v takto označených procedurách nelze volat jiné funkce a procedury, než ty označené k běhu na GPU a nelze použít rekurzi. Návratový typ CUDA kernelu musí být *void*.

V kernelech lze využít standardních programovacích technik jako například cyklů a větvení. Pro efektivní vykonání je nanejvýš vhodné se větvení vyhnout a snažit se kód kernelu navrhnout co nejpřímočařeji. Volání kernelu je od volání standardních metod odlišné. Musíme předem specifikovat v kolika vláknech se kernel spustí a specifikovat jejich rozložení mezi větší celky (gridy a bloky). Kernel se volá pomocí notace $<<< grid_dimension, block\_dimension >>>$ mezi názvem kernelu a jeho parametry.

Proměnné $grid\_dimension$ i $block\_dimension$ jsou proměnné typu $dim3$. v současných zařízeních mají 3 rozměry$(x, y, z)$. v případě $grid\_dimension$ se jedná o počet bloků v gridu, kde maximální hodnota jednotlivých souřadnic $x, y, z$ je omezena na 65535 bloků v každém směru. Dimenze $z$ v mřížce bloků je podporována až od GPU odpovídajících specifikaci výpočetních možností 2.0.

Proměnná $block\_dimension$ specifikuje počet vláken spuštěných v každém z těchto bloků. v závislosti na výpočetních možnostech karty je počet vláken v bloku omezen buď hodnotou 512, nebo 1024. Výsledný počet paralelně spouštěných kernelů (vláken) pak lze odvodit ze vztahu: $number\_of\_threads = grid\_dimension \times block\_dimension$.

```
//Definice CUDA kernelu
__global__ void kernel(){ \\... }

int main()
{
    // ...
    dim3 grid_dimension(10,10);
    dim3 block_dimension(256);
    kernel<<<grid_dimension, block_dimension>>>();
    // ...
}
```

Výpis 1: Ukázka definice a spuštění CUDA karnelu.

## 4.4 Hierarchie paměti

Vlákna spuštěná na GPU mají přístup k několika typům paměti. Ty se liší latencí – přístupovou dobou, velikostí a množstvím vláken, která k nim mají přístup. Obecně platí, čím je paměť větší, tím je pomalejší a dostupná více vláknům.

Typy pamětí řazeny podle přístupových dob od nejrychlejší:

- Registry
- Sdílená paměť
- Texturovací paměť
- Lokální paměť
- Globální paměť
- Konstantní paměť

### 4.4.1 Registr

Nejrychlejší a nejmenší paměť umístěná na samotném čipu. Skalární proměnné deklarované v kódu pro GPU bez speciálního identifikátoru bude uloženy právě do těchto registrů, které jsou navíc pro vyšší rychlost kešovány. Proměnná v registru je dostupná pro přístup i psaní pouze vláknu, ve kterém byla vytvořena a to pouze po dobu jeho běhu.

### 4.4.2 Lokální paměť

Využívá se pro pole proměnných v rámci vlákna. Je to nekešovaná paměť nacházející se mimo čip. Proměnné v této paměti jsou až 100x pomalejší než proměnné v registrech nebo sdílené paměti. Rozsah viditelnosti a délka životnosti proměnné v lokální paměti je stejná jako v registru, proměnná je dosažitelná pouze jedním vláknem po dobu jeho běhu.

### 4.4.3 Sdílená paměť

Rychlá, kešovaná paměť umístěna na čipu. Slouží pro ukládání sdílených proměnných, ale i polí proměnných. Sdílené proměnné jsou dostupné všem vláknům bloku, což sebou přináší možnost potenciální kooperace vláken při výpočtech. Životnost končí se skončením běhu všech vláken bloku. Vzhledem k její latenci může být sdílená paměť použita pro kešování zpracovávaných dat z Globální paměti. Je tedy obětována cena operace kopírování dat ve prospěch výrazně větší propustnosti, což se vyplatí především u četných operací s proměnnými uloženými v Globální paměti.

### 4.4.4 Globální paměť

Stejně jako lokální paměť je nekešovaná a řádově pomalejší než registry nebo sdílená paměť. Je mnohonásobně větší, dosahuje velikostí několika GB. Jedná se o hlavní paměť grafického adaptéru, do které jsou pomocí rozšiřujících funkcí nakopírována vstupní data pro další zpracování. Proměnné v této paměti setrvají po celou dobu běhu programu a jsou dostupné všem běžícím vláknům.

### 4.4.5 Konstantní paměť

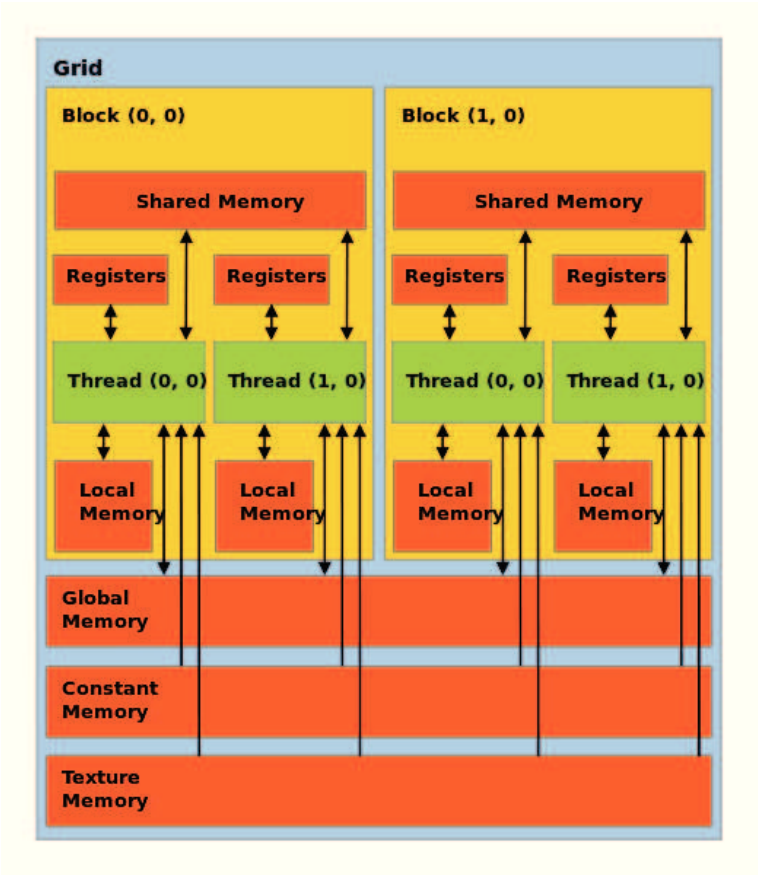
Je kešovaná paměť, určena pouze ke čtení po celou dobu běhu aplikace a to za možnosti přístupu všech vláken.

### 4.4.6 Texturovací paměť

Stejně jako konstantní paměť je kešovaná a určena pouze ke čtení po celou dobu běhu aplikace a to za možnosti přístupu všech vláken. Je optimalizovaná pro zpracování dvourozměrných dat, u kterých při čtení blízkých texturovacích souřadnic poskytuje lepší výkon, než v případě uložení těchto dat v konstantní paměti.

| Typ paměti | Deklarace proměnné | Viditelnost | Životnost |
|---|---|---|---|
| Registr | $int\ var$ | Vlákno | Vlákno |
| Lokální paměť | $int\ array\_var[10]$ | Vlákno | Vlákno |
| Sdílená paměť | $\_\_shared\_\_\ int\ shared\_var$ | Blok | Blok |
| Globální paměť | $\_\_global\_\_\ int\ global\_var$ | Grid | Aplikace |
| Konstantní paměť | $\_\_constant\_\_\ int\ constant\_var$ | Grid | Aplikace |
| Texturovací paměť | | Grid | Aplikace |

Tabulka 2: Hierarchie paměti.




Obrázek 4: Hierarchie paměti. [7]

# 5 Implementace

Praktická část této práce se zabývá efektivní implementací segmentační metody Blurring Mean-Shift s Gaussovým kernelem (GBMS) a jejím použitím k segmentaci obrazu. Algoritmus je implementován ve 2 verzích. Verze pro CPU využívá paralelizace pomocí technologie OpenMP. GPU implementace pracuje s masivním paralelismem využitím technologie CUDA.

## 5.1 Použité technologie

- Visual Studio 2010
- CUDA VS Wizard
- OpenCV 2.1
- OpenMP

### 5.1.1 CUDA VS Wizard

Microsoft Visual Studio nemá nativní podporu tvorby CUDA aplikací. Tuto funkcionalitu přináší CUDA VS Wizard, která VS nakonfiguruje tak, aby bylo možné vytvořit kompilovatelný projekt se všemi výhodami CUDY. Samozřejmě lze vývojové prostředí nakonfigurovat ručně, nicméně za podstatně delší čas.

### 5.1.2 Open MP

V dnešní době jsou běžné dvou i vícejádrové procesory CPU, nicméně při běžném návrhu program pracuje pouze s jádrem jedním, čímž se ochuzuje o potencionální výkonový nárůst. Open MP je multiplatformní aplikační rozhraní pro paralelní zpracování. Stěžejní část OpenMP jsou direktivy preprocesoru, určující například paralelizaci cyklu nebo rozvětvení programu do více vláken. Toto rozhraní je k dispozici jazykům C/C++ a Fortran.

## 5.2 O aplikaci

Program gbms.exe je konzolová aplikace poskytující funkcionalitu segmentace obrazu metodou Gaussian Blurring Mean-Shift se zadanými parametry. Podporovaná je především platforma Microsoft Windows 7.

Pro práci s obrazy, jejich načítání, ukládání i zobrazování je použita knihovna OpenCV. Formát vstupního obrazu je proto omezen výčtem podporovaných formátů funkce cvLoadImage z této knihovny. Mezi podporovanými formáty nechybí žádný z běžně podporovaných formátů, proto je pro použití v této aplikaci více než dostatečná. Testování probíhálo především na souborech typu .JPEG a .PNG. Všechny obrazy jsou prostřednictvím funkcionality OpenCV při načtení do aplikace převedeny na šedotónové.

Stávající verze systému Windows disponují mechanismem "Timeout Detection a Recovery"(TDR), který zajišťuje restart ovladače grafického ovladače při problémech. Tato vlastnost může mít neblahý vliv na aplikace, u kterých výpočetní doba překračuje stanovený limit. Systém takové výpočty automaticky ukončí a obnoví ovladač GPU. Pro správný běh aplikace GBMS je vhodné upravit systémové registry a TDR vypnout, popřípadě nastavit čas, po kterém se ovladač automaticky obnoví, na vyšší hodnotu.

Syntaxe:
gbms.exe -f source_image_path -i number_of_iteration [-c]
-f Povinný parametr následovaný systémovou cestou ke zdrojovému obrazu.
-s Povinný parametr určující hodnotu parametru $\sigma$.
-i Povinný parametr určující maximální počet iterací.
-c Volitelný parametr. Indikuje spuštění algoritmu pomocí CUDA technologie.

Vstupy aplikace:

- Obraz určený k segmentaci.
- Počet iterací algoritmu.
- Parametr SIGMA upřesňující tvar Gaussova kernelu.

Výstupy aplikace:

- Obraz zpracovaný metodou GBMS.

## 5.3 Popis běhu aplikace

Běh aplikace gbms.exe se dá rozdělit do několika logických celků:

1. Zpracování vstupních parametrů.
2. Vykonání výpočetního jádra GBMS.
3. Iterační smyčka.
4. Rekonstrukce obrazu.

### 5.3.1 Zpracování vstupních parametrů

Při startu aplikace se ověří správnost rozsahu vstupních parametrů, ověří se správnost cesty ke vstupnímu souboru. Dojde k načtení obrazu. Obraz je pro rychlé zpracování důležité reprezentovat co nejjednodušeji. GBMS algoritmus pro šedotónový obraz potřebuje k ztvárnění jednoho pixelu 3 proměnné, $[x, y, i]$. Je tedy jednoznačně určená poloha pixelu $x, y$ i jeho jas $i$. v aplikaci je zavedena nová struktura $Point$, která přesně takto pixel zobrazuje.

```
Struct Point{
   Float x; // Souřadnice x normovaná šířkou obrazu.
   Float y; // Souřadnice y normovaná výškou obrazu.
   Float z; // Hodnota jasu normovaná 256, tedy maximálním rozsahem hodnot.
}
```

Výpis 2: Definice struktury Point.

Všechny z vnitřních proměnných této struktury jsou přizpůsobeny pro co nejrychlejší výpočet. Proto jsou normovány a jejich hodnoty nabývají hodnot z intervalu $< 0, 1 >$. Pro výpočet GBMS je obraz reprezentovaný jako pole struktur $Point$ o rozměrech šířka×výška obrazu.

### 5.3.2 Vykonání výpočetního jádra GBMS

Obecně lze mean-shift algoritmy označit za velmi výpočetně náročné. Celkovou dobu běhu ovlivňuje velikost segmentovaných dat, v našem případě rozměry vstupního obrazu, ale velmi výrazně ji ovlivňuje velikost výpočetního okna, v rámci něhož je každý bod zpracován. Kernely s omezeným definičním oborem hodnot $D(f)$ lze velikosti okna přizpůsobit tak, aby krajní hodnoty intervalu definičního oboru odpovídaly krajním hodnotám výpočetního okna. Tento fakt však nelze aplikovat na Gaussův kernel, který má $D(f) = R$. Velikost výpočetního okna je proto v případě použití Gaussova kernelu rovna rozměrům celého obrazu. Toto samozřejmě platí i pro naše použití v algoritmu GBMS, kde je Gaussův kernel $K(x)$ definován rovnicí 5.

$$K(x) = e^{-\frac{1}{2}\frac{x^2}{\sigma^2}} \tag{5}$$

Kde $x$ je euklidovská vzdálenost bodů v daném prostoru $[x, y, z]$ a $\sigma$ je parametr určující přesný tvar kernelu.

Je tedy zřejmé, že změnou parametru sigma výpočet neurychlíme, pouze změníme podobu výsledků tím, že se budou pixely do výpočtu započítávat jinou vahou. Čím menší sigma je voleno, tím "přísnější" tato funkce je a výsledné segmenty menší. Velké množství malých segmentů není pro segmentaci žádoucí, proto je nutné volit vhodnou hodnotu tohoto parametru. z definice Gausova kernelu lze určit, že od určitých vzdáleností x nám tato funkce vrací hodnoty velmi blízké nule, takové body pak obraz ovlivňují velmi malou vahou v poměru k časové náročnosti jejich zpracování. Pro korektnost a porovnání výsledků je počítáno i s těmito hodnotami. Pro výrazné urychlení aplikace by bylo vhodné tyto hodnoty zanedbat. v aplikaci je Gaussův kernel reprezentován funkcí $gaussianF$.

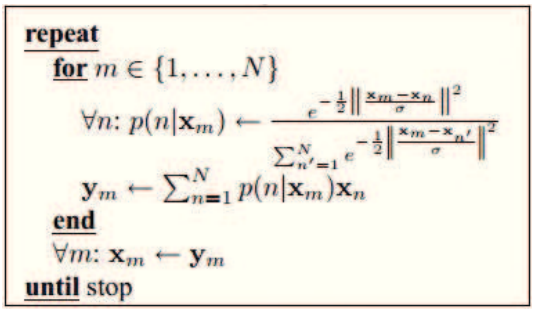
Funkce $gaussianF$ vrací hodnotu reprezentující váhu pixelu ve výpočtu v závislosti na vstupních parametrech distanceSqr, tedy druhé mocnině vzdálenosti pixelu a parametru sigma zmíněného výše.

Další často používanou proměnnou je vzdálenost dvou bodů v prostoru. Je nutná jako vstupní parametr pro výpočet váhy pomocí Gaussova kernelu. Obecně je vzdálenost bodů $A$ a $B$, často také označována jako velikost úsečky $|AB|$ definována vztahem 6.

$$|AB| = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + (b_3 - a_3)^2} \tag{6}$$

Tento vzorec lze snadno aplikovat na dva body našeho typu $Point$. Pro naše účely, kdy ve většině případů nepotřebujeme vzdálenost, ale její druhou mocninu (viz. definice Gaussova kernelu), odpadá nutnost odmocnění, což příznivě ovlivní čas výpočtu. v aplikaci je tedy definovaná funkce $distanceSqr$ vracející tuto hodnotu.

Pomineme-li smyčku iterační, samotné jádro GBMS algoritmu se skládá ze 2 vnořených cyklů. První, vnější, prochází jednotlivé body vstupních dat. v druhém, vnitřním, probíhá samotný výpočet nové pozice odpovídajícího bodu. Tento cyklus tedy prochází body odpovídajícího výpočetního okna a počítá jejich vážený průměr na základě jejich euklidovské vzdálenosti od prvku, kterému počítá novou pozici. Tato vzdálenost je pak vahou s jakou jednotlivé prvky výpočetního okna ovlivní jednotlivé posuny. Na obrázku 5 je algoritmus GBMS zapsán v pseudokódu.




Obrázek 5: GBMS algoritmus zapsán v pseodokódu. [1]

Smyčka $repeat - until$ ve výše zmíněném zápisu 5 představuje jednotlivé iterace výpočtu. Vnější cyklus $for\ m \in \{1, \ldots, N\}$ prochází všechny datové body, vnitřní smyčka zpracovává jejich odpovídající výpočetní okna a postupně určuje jejich novou pozici.

Vzhledem k faktu, že výpočetní jádro se skládá ze dvou vnořených cyklů, z nichž každý se pro obraz s $n$ pixely provede $n$ krát. Výpočetní složitost GBMS nad obrazem s $n$ pixely je $O(n^2)$.

### 5.3.3 Iterační smyčka

Po dobu celého výpočtu se pracuje nad dvěma poli, reprezentující vstupní a výstupní datovou množinu v dané iteraci. Po skončení iterace se všechny body výstupního pole zkopírují do pole vstupního a pokud není splněna podmínka ukončení výpočtu, algoritmus proběhne znovu nad touto novou množinou. Kernely pracující s CPU mají data reprezentující obraz uložena klasicky v paměti RAM. Jejich kopírování mezi iteracemi zprostředkovává metoda $void\ copyArr(...)$. CUDA verze algoritmu má tato data uložena v globální paměti grafického adaptéru. Využíváme tedy metod ke kopírování dat na GPU.

### 5.3.4 Rekonstrukce obrazu

Po skončení celého výpočtu je potřeba převést celý obraz zpět do datového typu zobrazitelného na obrazovce pomocí standardních funkcí knihovny OpenCV.

# 6 Výpočetní jádra

Výpočetním jádrem aplikace rozumíme část kódu vykonávající jednu iteraci GBMS nad celým obrazem. Ve všech případech je výpočetním jádrem funkce typu void s parametry ukazatel na zdrojové a cílové pole typu Point, šířka a výška obrazu. v případě CUDA implementace výpočetním jádrem rozumíme samotný CUDA kernel.

Při vývoji bylo implementováno několik verzí takovýchto jader. v další kapitole jsou pak porovnány jejich výkony a náročnosti implementace.

## 6.1 CPU

Původní implementovaná verze GBMS algoritmu nevyužívá žádnou z výhod paralelního zpracování. Výpočetní okna všech bodů vstupního obrazu jsou zpracovávána sériově jedním procesorovým jádrem. Toto výpočetní jádro je dále v textu a výsledcích označováno jako $CPUkernel$. Poměrnou snadnost implementace takového jádra pak převyšují nevýhody, především vysoký výpočetní čas.

```
void CPUkernel1(Point* srcArr, Point* dstArr,int IMG_HEIGHT,int IMG_WIDTH, float sigma)
{
    float sw, sx, sy, sz, w;
    for(int i=0;i<IMG_HEIGHT*IMG_WIDTH;i++)
    {
        sw = sx = sy = sz = w = 0;
        for(int j=0;j<IMG_HEIGHT*IMG_WIDTH;j++)
        {
            w = gaussianF(distanceSqr(srcArr[i],srcArr[j]),sigma);
            sx += w*srcArr[j].x;
            sy += w*srcArr[j].y;
            sz += w*srcArr[j].z;
            sw += w;
        }
        dstArr[i].x = sx/sw;
        dstArr[i].y = sy/sw;
        dstArr[i].z = sz/sw;
    }
}
```

Výpis 3: CPU kernel 1.

### 6.1.1 Paralelizace využitím OpenMP

I přes to, že současné procesory CPU nedisponují takovým výpočetním výkonem jako GPU, dá se výsledná efektivita a rychlost programu pro CPU značně urychlit využitím vícejádrových procesorů. Nezávislé výpočty se vykonávají v oddělených vláknech na jednotlivých jádrech procesoru. Algoritmus je nakonfigurován tak, že se automaticky spouští na všech dostupných procesorových jádrech včetně virtuálních v případě technologie HyperThreading. Výsledky výpočetního jádra $CPUkernel$ jsou tedy doplněny o informaci

na kolika procesorových jádrech byl spuštěn. Naměřené časy běhu pak ukazují jak přínos dalších výpočetních jader, tak přínos technologie HyperThreading. Paralelizace smyček pomocí OpenMP sebou přináší nutnost správně specifikovat viditelnost jednotlivých proměnných v rámci paralelních větví tak, aby nedocházelo ke konfliktům a nekonzistenci dílčích výsledků. Výpis kódu 4 demonstruje konkrétní použití direktivy OpenMP pro implementovaný GBMS algoritmus.

```
void CPUkernel1(Point* srcArr, Point* dstArr,int IMG_HEIGHT,int IMG_WIDTH, float sigma)
{
    float sw, sx, sy, sz, w;
    int i,j;
    #pragma omp parallel for private(i,j,w,sx,sy,sz,sw) shared(dstArr,srcArr,IMG_HEIGHT,
        IMG_WIDTH,sigma)
    for(i=0;i<IMG_HEIGHT*IMG_WIDTH;i++)
    {
        // ...
    }
}
```

Výpis 4: Použití direktivy OMP.

## 6.2 GPU

V rámci vývoje bylo implementováno několik GPU kernelů, které využívají různou šíři paralelizace jak v celkovém počtu paralelně spuštěných vláken, tak v jejich virtuálním rozdělení do bloků. Různými přístupy k použití sdílené paměti byla optimalizována rychlost přístupu k potřebným datům a práce s nimi. Následující kapitoly popisují rozdílně implementované GPU kernely.

### 6.2.1 GPU Kernel 1

Toto výpočetní jádro paralelizuje pouze vnější cyklus procházející jednotlivé body obrazu tím, že spustí pro výpočet nové pozice bodu jedno vlákno. Celkově je tedy tento kernel pro obraz o $n$ bodech spuštěn v $n$ paralelních vláknech. Ideální rozmístění vláken z pohledu univerzálního nasazení tohoto kernelu je použít šířku a výšku obrazu jako jednotlivé počty bloků v gridu. v takovém případě je velikost vstupního obrazu v pixelech limitována maximálními rozměry gridu, tedy 65000x65000, což je pro většinu současných obrazů dostačující. v kapitole zabývající se výkonem je znázorněn vliv různých rozložení vláken do bloků na celkový čas výpočtu. Cyklus zabývající se výpočtem nové pozice bodu v rámci výpočetního okna zůstává sériový. Každé spuštěné vlákno zpracuje celé výpočetní okno, všechny proměnné potřebné pro výpočet nové pozice bodu jsou uloženy v rychlých registrech.

```
__global__ void GPUkernel1(Point* srcArr, Point* dstArr, int width, int height, float sigma)
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
```

```
    float sw, sx, sy, sz, w;
    sw = sx = sy = sz = w = 0;
    for(int j=0;j<width*height;j++)
    {
        w = gaussianF(distanceSqr(srcArr[i],srcArr[j]),sigma);
        sx += w * srcArr[j].x;
        sy += w * srcArr[j].y;
        sz += w * srcArr[j].z;
        sw += w;
    }
    dstArr[i].x = sx/sw;
    dstArr[i].y = sy/sw;
    dstArr[i].z = sz/sw;
}
```

Výpis 5: GPU kernel 1.

### 6.2.2 GPU Kernel 2

Druhé implementované a testované výpočetní jádro pro GPU je v aplikaci reprezentováno pod názvem $GPUkernel2$. Tento kernel nevyužívá o nic větší paralelizace než kernel $GPUkernel1$. Každé výpočetní okno je zpracováno pouze jedním vláknem, s tím rozdílem, že v rámci jednoho bloku je spuštěno 256 vláken, aby mohla přistupovat k jedné sdílené paměti. Body aktuálně se podílející na výpočtu jsou napřed načteny z globální do sdílené paměti a až pak jsou zpracovány. Pole ve sdílené paměti tedy tvoří keš aktuálně zpracovávaných bodů v rámci bloku. Z důvodu omezení konfliktů při kopírování dat ze sdílené paměti, vlákna každého bloku začínají zpracovávat výpočetní okno z jiného místa. Řešit kolize v rámci čtení všech vláken jednoho bloku nemusíme, jelikož CUDA dokáže uzamknout část paměti do tzv. $broadcast$ režimu, ve kterém je takovéto čtení dokonce rychlejší.

```
__global__ void GPUkernel2(Point* srcArr, Point* dstArr, int width, int height, float sigma)
{
    int index = blockIdx.x*blockDim.x+threadIdx.x;
    // ...
    Point currentPoint = srcArr[index];
    __shared__ Point sharedArr[BLOCK_DIMENSION];

    for(int k~= 0; k~< gridDim.x ;k++)
    {
        sharedArr[threadIdx.x].x = srcArr[(threadIdx.x+blockDim.x*k)%(width*height)].x;
        // ...
        __syncthreads();

        for(int i~= 0; i<blockDim.x; i++)
        {
            w = gaussianF(distanceSqr(currentPoint,sharedArr[i]),sigma);
            sx += sharedArr[i].x * w;
            // ...
```

```
            sw+=w;
        }
    }
    dstArr[index].x = sx/sw;
    // ...
}
```

Výpis 6: GPU kernel 2.

### 6.2.3 GPU Kernel 3

V obou předchozích kernelech byl paralelizován pouze cyklus procházející jednotlivé body obrazu. Kernel $GPUkernel3$ paralelizuje i cyklus procházející jednotlivé body výpočetního okna. Vzhledem k potřebě dat z celého výpočetního okna pro výpočet nové pozice bodu je míra paralelizace tohoto cyklu omezena maximálním počtem vláken v bloku. Dodržení podmínky, že jedno výpočetní okno je zpracováno v rámci bloku, nám umožňuje efektivně využít sdílenou paměť pro uložení mezivýsledků zpracovaných jednotlivými vlákny bloku. z pohledu maximální šíře paralelizace by bylo ideální zpracovat každý bod výpočetního okna samostatným vláknem. Gaussův kernel takovou míru paralelizace v praxi neumožňuje, neboť se jeho výpočetní okno zvětšuje společně s velikostí vstupního obrazu. Počet bodů výpočetního okna v praktickém použití GBMS ve většině případů přesahuje maximální možný počet vláken spuštěných v bloku. v konečné verzi programu je každé výpočetní okno zpracováno konstantním počtem vláken 256. Tento kernel je tedy v praxi pro obraz s $n$ pixely spuštěn v $n$ blocích, reprezentující všechny výpočetní okna, z nichž každé je zpracováno 256 vlákny.

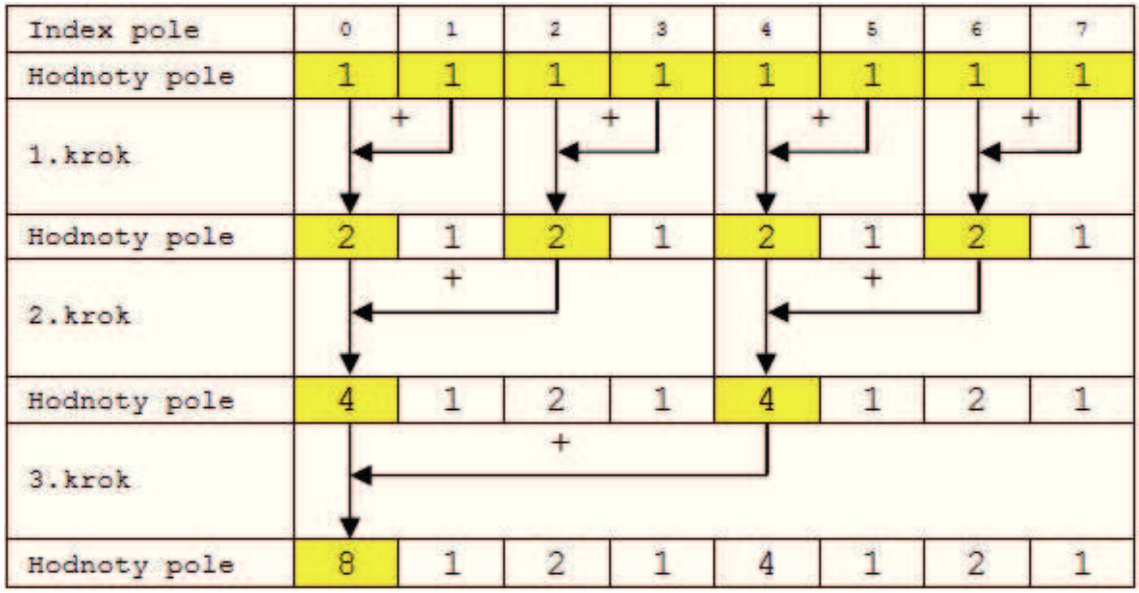
```
__global__ void GPUkernel3(Point* srcArr, Point* dstArr, int width, int height, float sigma)
{
    int index = blockIdx.x+blockIdx.y*gridDim.x;
    // ...
    Point currentPoint=srcArr[index];
    __shared__ Point sharedArr[BLOCK_DIMENSION];
    __shared__ float swSharedArr[BLOCK_DIMENSION];

    for(int k= 0;k<width*height/BLOCK_DIMENSION;k++ )
    {
        w = gaussianF(distanceSqr(currentPoint,srcArr[threadIdx.x+BLOCK_DIMENSION*k]),
            sigma);
        sw+=w;
        sx+=srcArr[threadIdx.x+BLOCK_DIMENSION*k].x*w;
        // ...
    }
    swSharedArr[threadIdx.x] = sw;
    sharedArr[threadIdx.x].x = sx;
    // ...
    //Paralelizovany soucet hodnot pole SharedArr.x, .y, .z a swSharedArr
    // ...
}
```

Výpis 7: GPU kernel 3.

Rozdělení zpracování výpočetního okna mezi více vláken s sebou přináší nutnost zpracování jednotlivých mezivýsledků uložených v poli ve sdílené paměti. Obecně je velikost tohoto pole stejná jako počet vláken v bloku, aby každé vlákno mělo místo, kde může uložit svůj mezivýsledek. Při standardním zpracování pole o velikosti 256 potřebujeme ke zpracování 256 průchodů cyklem. i tento cyklus lze v CUDA aplikaci efektivně paralizovat.

Paralizace zpracování hodnot sdíleného pole: Mějme pole čísel o velikosti $n$. v prvním kroku paralelně přičteme k prvkům na sudých pozicích prvky na pozici o vzdálenost 1 větší. v dalších krocích se tato vzdálenost zvětšuje a opět dochází k sečtení. Tímto systémem jsme schopni sečíst pole o velikosti $n$ v $log_2(n)$ krocích. Potřebných 256 prvků tedy zpracujeme v 8 krocích, což je výrazné urychlení oproti původním 256 krokům.




Obrázek 6: Princip paralelního součtu hodnot pole.

# 7 Dosažené výsledky

Všechna výpočetní jádra dosahují při shodné parametrizaci totožných výsledků. Na přiloženém obrázku 7 lze vidět výsledky segmentace algoritmem GBMS pro různý počet proběhlých iterací. Referenční obrázek je o velikosti 128x128 pixelů a parametr $\sigma = 0.05$. Na první pohled je patrný úbytek segmentů se zvyšujícím se číslem iterace.

Obrázek 7: Průběh GBMS v jednotlivých iteracích.

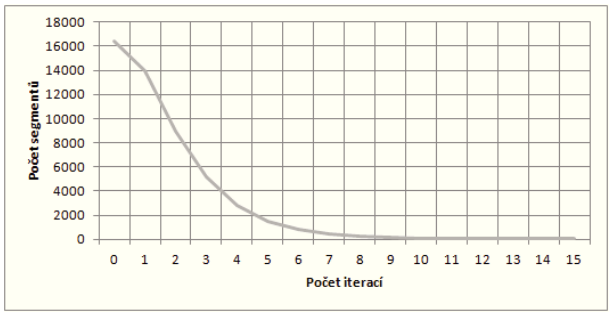
Počet segmentů je měřen jako počet rozdílných atraktorů, ke kterým jednotlivé body obrazu postupným posunem doputovaly. Závislost počtu výsledných segmentů na počtu proběhlých iterací nám dokazuje graf na obrázku 8, který vychází z hodnot naměřených v tabulce 3. z těchto měření je také jasně patrná konvergence výpočtu nastávající po 15 iteracích.

| Počet iterací | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Počet segmentů | 16384 | 13949 | 8973 | 5177 | 2811 | 1505 | 832 | 471 |
| Počet iterací | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Počet segmentů | 250 | 150 | 109 | 78 | 62 | 47 | 38 | 37 |

Tabulka 3: Závislost počtu segmentů na počtu iterací.

Dalším zajímavým měřením je vztah parametru $\sigma$ a počtu segmentů při stejném počtu iterací výpočtu. Tabulka 4 a z ní vycházející graf na obrázku 9 potvrzují, že s rostoucí hodnotou $\sigma$ počet segmentů klesá.

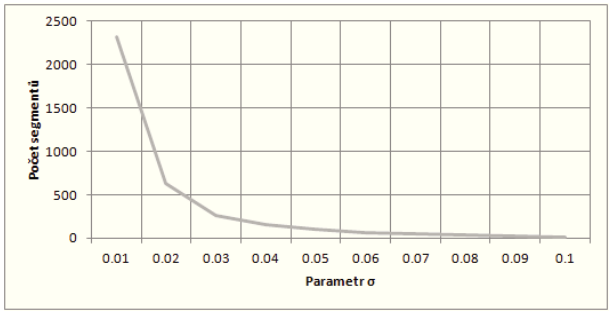
Z těchto poznatků můžeme usoudit, že vhodnou volbou $\sigma$ parametru můžeme dojít k značně podobným výsledkům v podstatně odlišných časech tím, že algoritmus provede různý počet iterací. Nelze však volit velké hodnoty $\sigma$ bez ohledu na to, jak velký detail jednotlivých segmentů potřebujeme zachovat. Přehnaně velké hodnoty tohoto parametru pak způsobí, že výsledný obraz se "slije"do malého počtu segmentů v krátkém čase, nicméně tyto segmenty jsou nicneříkající vzhledem k původnímu obrazu.




Obrázek 8: Závislost počtu segmentů na počtu iterací.

| Parametr $\sigma$ | 0.02 | 0.03 | 0.04 | 0.05 | 0.07 | 0.08 | 0.09 | 0.1 |
|---|---|---|---|---|---|---|---|---|
| Počet segmentů | 631 | 255 | 155 | 109 | 67 | 54 | 32 | 20 |

Tabulka 4: Závislost počtu segmentů na parametru $\sigma$.




Obrázek 9: Závislost počtu segmentů na parametru $\sigma$.

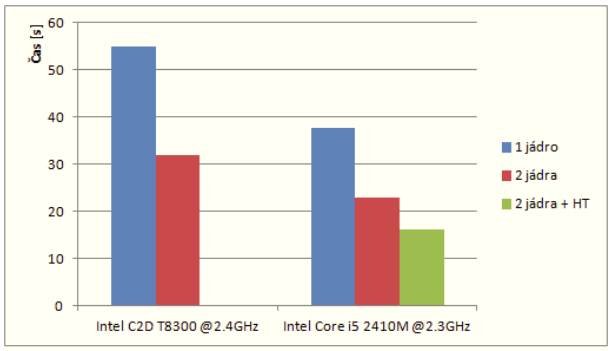
Kritérium zastavení výpočtu na základě porovnání hodnot entropií histogramu se ukázalo jako funkční a zastavilo výpočet ve chvíli, kdy už se vzhled výsledných segmentů nemění a obraz je dostatečně segmentován.

Všechna tato měření potvrzují teoretický základ GBMS algoritmu a přesvědčují nás o správnosti implementovaných řešení.

## 7.1 Výkon

Klíčovou hodnotou pro porovnání výkonu jednotlivých kernelů je doba jejich běhu. Měřila se výhradně doba vykonávání samotného výpočetního jádra včetně operací kopírování dat z RAM do DRAM a naopak. Do času tak nejsou započítány operace inicializace, převod a zobrazení obrazu. v konečném důsledku by tyto časy byly zanedbatelné a zbytečně zkreslovaly celkovou dobu výpočtu samotného GBMS algoritmu. Všechna výpočetní jádra dosahují stejného výsledku ve stejném počtu vykonaných operací, široce se však liší v dosažených výkonech.

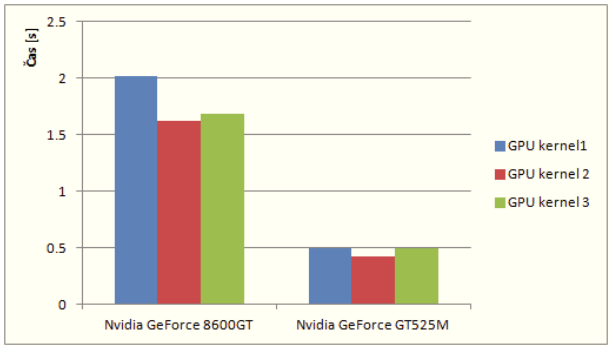
Měření byla provedena na procesorech Intel C2D T8300 @2.4GHz, Intel Core i5 @2.3GHz a grafický adaptérech NVIDIA GeForce 8600GT a GeForce GT525M. Měření nás jednoznačně přesvědčila o výkonnostní převaze masivního paralelního zpracování pomocí GPU.




Obrázek 10: Výsledné doby běhu jedné iterace GBMS na CPU pro obraz 128x128 pixelů.

| | 1 jádro | 2 jádra | 2 jádra + HT |
|---|---|---|---|
| Intel C2D @2.4GHz | 54.87 | 31.96 | - |
| Intel Core i5 @2.3GHz | 37.79 | 22.8 | 16.08 |

Tabulka 5: Doba běhu CPU kernelů.




Obrázek 11: Výsledné doby běhu GPU kernelů pro obraz 128x128 pixelů.

| | GPU kernel 1 | GPU kernel 2 | GPU kernel 3 |
|---|---|---|---|
| NVIDIA GeForce 8600GT | 2.02 | 1.62 | 1.68 |
| NVIDIA GeForce GT525M | 0.50 | 0.42 | 0.49 |

Tabulka 6: Doba běhu GPU kernelů [s].

Urychlení GBMS algoritmu pomocí zpracování grafickým adaptérem je v řádech desítek procent. Přesně naměřené výsledky jsou znázorněny grafy na obrázku 10 a 11.

Nemůžeme opomenout přínos technologie OpenMP, která u obou dvoujádrových procesorů přináší téměř poloviční čas výpočtu zapojením druhého jádra. Procesor Intel Core i5 podporuje technologii Hyper-threading, která z jednoho fyzického procesoru vytváří 2 virtuální. v systému se tedy tento procesor chová jako čtyřjádrový. OpenMP na tomto procesoru tedy dokáže algoritmus GBMS paralelizovat jako pro procesor čtyřjádrový. Výkonový nárůst s použitím Hyper-threadingu je opět velice patrný, nicméně už ne tak výrazný, jako bylo přidání fyzického procesorového jádra.

Jak je patrné z grafu na obrázku 11, jednotlivé implemetace GPU kernelů se mezi sebou výkonově liší také a to úsporou času až 20% při vykonání jedné iterace GBMS. $GPUkernel1$ jedním vláknem zpracovává celé výpočetní okno, obsahuje tedy cykly o délce velikosti výpočetního okna. Nicméně toto jedno vlákno má k dispozici mezivýsledky uložené v rychlých registrech a odpadá nutnost režie více vláken podílejících se na zpracování jednoho výpočetního okna. Paralelizace u tohoto kernelu dosahuje hodnot, ve kterých je schopen efektivně využít potenciál grafického čipu. Obecně však tuto implementaci můžeme odsoudit za její špatný potenciál škálovatelnosti a celkově nejhorší naměřené výkony mezi GPU kernely.

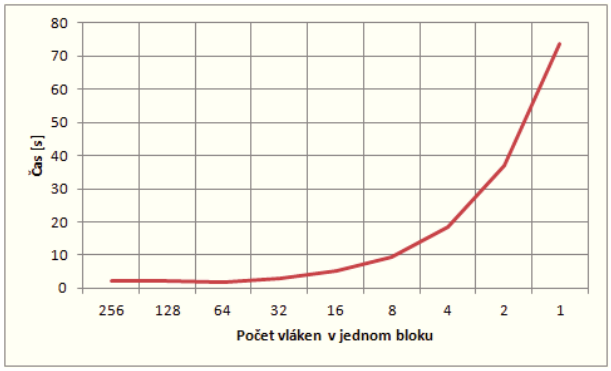
$GPUkernel2$ se naopak ve většině testů projevil nejlépe. Paralelizace problému je v naprosto stejné míře jako u kernelu předcházejícího. Vylepšení spočívá v "cachování"množin aktuálních bodů výpočetního okna do sdílené paměti. Při operacích měřící vzdálenost bodů a jejich váhy tedy pracujeme s takto bufferovanými hodnotami. Dále jsou pak minimalizovány vzniklé konflikty při samotném bufferování z globální paměti. Vlákna každého bloku začínají zpracovávat výpočetní okno v jiné části vždy posunuté o velikost bloku.

Posledním implementovaným kernelem je $GPUkernel3$. Využívá největší šíře paralelizace, kde každé výpočetní okno je zpracováno 256 vlákny ukládající své dílčí výsledky do sdílené paměti, jejíž prvky jsou sečteny technikou paralelních součtů. Délka vykonání kernelu je tedy zkrácena o nemálo průchodů vnitřních cyklů.

Analýza kernelů programem CUDA Visual Profiler ukázala úzké hrdlo implementace v podobě nízkého počtu registrů adaptéru NVIDIA GeForce 8600GT související s podporou pouze "CUDA Capabilities 1.2". i přes snížení počtu proměnných uložených v registrech na minimum, jsou právě ony důvodem k neúplnému vytížení jednotlivých multiprocesorů. Lze očekávat, že na modernějších grafických adaptérech podporující specifikaci výpočetní možnosti 2.0 a vyšší, bude vytížení grafického čipu ještě efektivnější.

Ve všech dosavadních testech jednotlivých implementovaných kernelů byl přínos technologie CUDA v řádech desítek procent v závislosti na tom, s jakým procesorem byly časy porovnány. Neznamená to však, že jakkoliv implementovaný a parametrizovaný CUDA kernel přinese takovéto zrychlení výpočtu. Nevhodnou parametrizací spouštění kernelu a tím nedostačného vytížení GPU může nastat situace, kdy CPU dosáhne lepších výsledků. Tento fakt se nejlépe demonstruje na výpočetním jádře GPU 1, které je vždy spuštěno stejným počtem vláken odpovídající počtu pixelů v obraze a výpočetní okno je vždy zpracováno pouze jedním vláknem sekvenčně v cyklu. Liší se však rozmístěním

vláken v blocích. v extrémních případech tedy můžeme výpočet tímto kernelem spustit tak, že každému pixelu odpovídá jeden blok o jednom vláknu nebo ho spustit v x-krát méně, x-krát větších blocích.




Obrázek 12: Vliv dimenze bloků na celkový čas vykonání jedné iterace GBMS nad obrazem 128x128. Počet spuštěných bloků je pak dán vztahem $128 * 128/dimenze$.

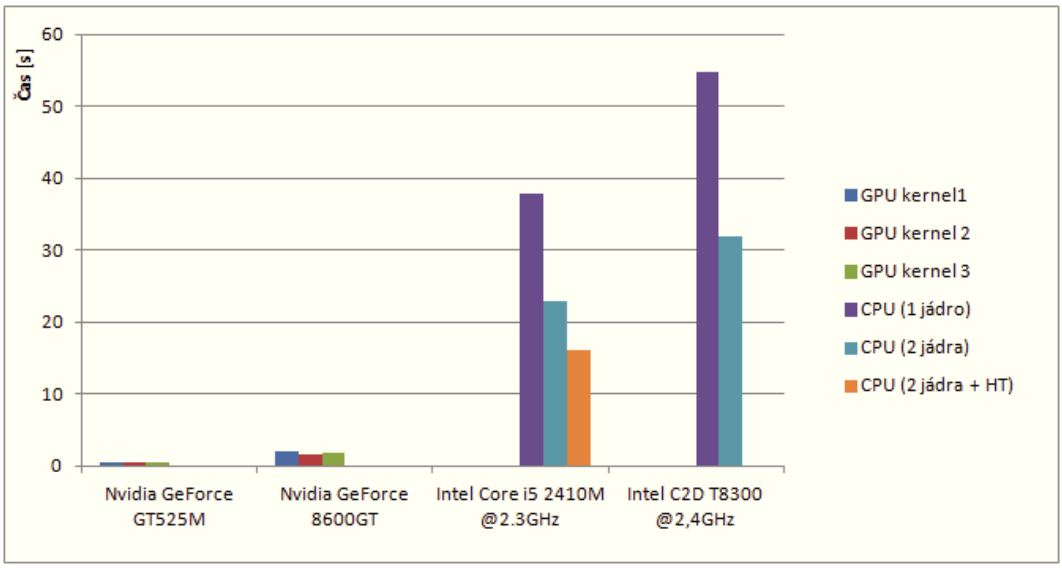
Lineární časový nárůst můžeme pozorovat až po překročení práhu 64 vláken / blok. Tato hodnota zřejmě souvisí s HW architekturou GPU, pokud tedy snížíme dimenzi bloku pod tuto mez, nebude využit výpočetní potenciál multiprocesorů a dojde k nárustu času potřebného k vykonání iterace. v nejhorším případě byl výkon dokonce horší, než při zpracování jedním CPU jádrem.

Jedním z podstatných cílů této práce bylo demonstrovat přínos technologie CUDA ve srovnání s klasickými procesory. Graf na obrázku 13, vycházející z tabulek 5 a 6, lineárním měřítkem porovnává časy potřebné k vykonání jedné iterace GBMS nad vstupem o velikosti 128×128 pixelů. Lineární měřítko je použito pro jasnou představu o výkonnostních rozdílech jednotlivých implementací.

Měření jasně prokázalo, že v nejlepším případě je urychlení algoritmu více než 100 násobné. Čas potřebný k vykonání algoritmu na GPU Nvidia GeForce GT525M je dokonce až 130× nižší než na CPU Intel C2D (bez použití OMP).

Srovnání časů na současném procesoru Intel Core i5 s časy několik let staré GeForce 8600GT opět vyznělo ve prospěch CUDA GPU. v případě použití 1 procesorového jádra Core i5 je CUDA implementace 23× rychlejší. Dokonce ani při využití všech výpočetních jader včetně technologie HT nepředstihl tento procesor zastaralou GeForce 8600GT. v tomto případě byl algoritmus na GPU rychlejší 10×.

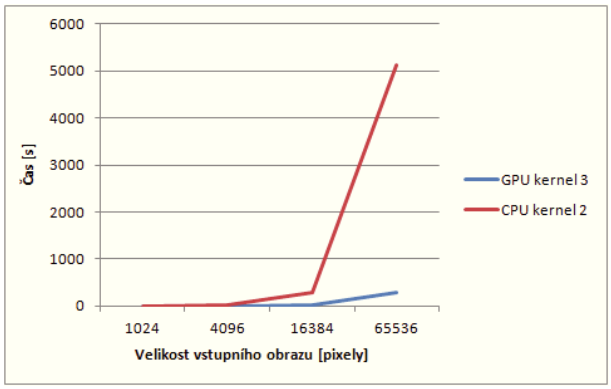
Nejobjektivnější pohled na zrychlení CUDA implementace poskytuje srovnání dosažených časů na HW vyrobeném ve stejné "technologické generaci".V našem případě takto můžeme srovnat GPU GeForce 8600GT s procesorem Intel C2D a GeForce GT525M s Intel Core i5. v prvním jmenovaném případě je algoritmus na GPU rychlejší 33×, v druhém 90×, rovněž ve prospěch grafického adaptéru. Větší urychlení algoritmu v druhém případě souvisí s podporou specifikace "CUDA Compute Capabilities"ve verzi 2.1, což s sebou přináší mimojiné 4× větší počet registrů v každém multiprocesoru této grafické




Obrázek 13: Porovnání všech implementací na jednotlivých zařízeních.

karty. Právě jejich malý počet byl hlavní překážkou pro spuštění většího počtu vláken na jednotlivých multiprocesorech adaptéru GeForce 8600GT.

Je patrné, že výkonový nárůst je silně závislý na generaci a typu použitého HW. i přesto, že se výrobci procesorů stále častěji uchylují k výrobě vícejádrových procesorů, které na trhu určitě mají své místo, jejich výkon v paralelelních úlohách značně pokulhává za stávajícími GPU. Což jsme také dokázali ve výše uvedených testech.




Obrázek 14: Závislost výpočetního času na velikosti vstupního obrazu. Výpočet 10 iterací GBMS pomocí NVIDIA GeForce 8600GT a Intel C2D T8300 (2 jádra).

Měření znázorněno grafem na obrázku 14 ukazuje závislost času potřebného k výpočtu 10 iterací GBMS nad různě velkými vstupními obrazy. Tento graf opět používá lineární měřítko z důvodu větší názornosti demonstrovaných výsledků. Už při poměrně malých vstupních obrazech je rozdíl výsledných časů nezanedbatelný a při jejich zvětšování

tato hodnota propastně narůstá. Dokonce i s použitím OpenMP roste výrazně rychleji, než výsledný čas hodnocené CUDA implementace. Čas věnovaný do návrhu a implementace CUDA kernelu se nepochybně vyplatí, zvláště při zpracování velkých vstupních dat.

V současné době firma NVIDIA představila novou architekturu GPU pod názvem ”Kepler”jako přímého nástupce architekury ”Fermi”. GPU založené na nové architektuře podporují ”CUDA Compute Capabilities 3.0”. Jednotlivé multiprocesory těchto GPU čítají 192 fyzických výpočetních jader, což sebou přináší velký potenciál k dalšímu výkonovému růstu.

# 8 Závěr

Tato práce nás seznamuje se základními metodami zpracování a segmentace obrazu. Konkrétně se pak zabývá clusterovacím algoritmem Mean-Shift a jeho modifikací Gaussian Blurring Mean-Shift. Algoritmus GBMS je implementován za pomocí technologií CUDA a OpenMP, které na něm demonstrují svůj přínos v oblasti rozsáhlých výpočtů. V jednotlivých kapitolách se pak věnuji teoretickému základu těchto algoritmů a popisu CUDA technologie. Praktická část práce se zabývá implementací algoritmu GBMS včetně návrhů několika výpočetních jader na základě získaných teoretických zkušeností.

Další kapitola této bakalářské práce se věnuje výsledkům implementovaných kernelů a jejich porovnání. Přesvědčili jsme se o výrazném nárůstu výkonu použitím technologií OpenMP a CUDA, u které byl výkonový nárůst nejpatrnější. Testy prokázaly, že využitím více procesorových jader včetně těch virtuálních, můžeme výrazně akcelerovat aplikaci i při běhu na CPU.

Práce také ukazuje, že špatnou parametrizací volání CUDA kernelu můžeme docílit dokonce horších výkonů, než u zpracování obrazu klasickým procesorem. Což nás utvrdilo o nutnosti nejen navrhnout funkční kernel, ale také ho výrazně optimalizovat.

Ve všech uskutečněných výkonových testech zvítězila navrhnutá CUDA implementace nad CPU implementací a to dokonce i v případě porovnání současného CPU s několik generací starším grafickým adaptérem. v nejlepším případě pak masivní paralelismus přinesl více než 100× rychlejší vykonání GBMS algoritmu. Vzhledem k výsledným časům běhu GBMS nad různými vstupy je každé urychlení algoritmu velkým přínosem, zvláště pak, když je výkonový nárůst takto markantní.

V průběhu implementace se vyskytlo několik problému souvisejících s vývojem CUDA aplikace. Jedním z nich byla nutnost omezit funkci TDR v systému Windows 7 pro správný běh aplikace. Je pravděpodobné, že takovéto nastavení registrů bude nutné provést na všech cílových stanicích.

Samotný GBMS algoritmus v praxi prokázal výborné segmentační vlastnosti v závislosti na nastavených parametrech. z mého pohledu největší síla této segmentační metody tkví v možnosti zpracovávat i obecná mnohorozměrná data s nepříliš velkým růstem složitosti výpočtu v závislosti na dimenzi dat. Na druhou stranu je to právě výpočetní složitost GBMS algoritmu, která zabraňuje jeho širšímu použití například u úloh určených ke zpracování v reálném čase. Použitý "neořezaný"Gaussův kernel sice obecně přináší výborné výsledky segmentace, avšak s ohledem na velikosti vstupních dat by v praktickém nasazení algoritmu bylo vhodné obětovat malou část přesnosti výsledku ve prospěch rychlosti jeho dosažení a body od určité vzdálenosti do výpočtu vůbec nezahrnout.

Implementace poskytuje další prostor k návrhů nových výpočetních kernelů i testování a optimalizaci těch stávajících. v současné době se s příchodem nových grafických adaptérů NVIDIA, založených na architektuře "Kepler", přímo nabízí provést optimalizaci a testy právě na těchto GPU.

Tato bakalářská práce je mým prvním projektem zabývající se technologií CUDA. Vzhledem k zajímavým dosaženým výsledkům a potencionálnímu výkonovému růstu v budoucnu věřím, že zdaleka ne projektem posledním.

# 9 Reference

[1] Carreira-Perpinán, M. Á. Fast Nonparametric Clustering with Gaussian Blurring Mean-Shift. *Proceeding ICML '06 Proceedings of the 23rd international conference on Machine learning.* 2006, 23, s. 153-160.

[2] COMANICIU,Dorin; MEER,Peter. Mean Shift: A Robust Approach Toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Inteligence.* May 2002, 24, 5, s. 603-619.

[3] COMANICIU,Dorin; MEER,Peter. *Mean Shift Analysis and Applications*. 1999.

[4] CHENG, Y. Mean Shift, Mode Seeking, and Clustering.*IEEE Trans. Pattern Anal. Machine Intel l.* 1995, vol. 17, 790-799.

[5] Kernel (statistics) In: *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 5.1.2007. Dostupné z: <http://en.wikipedia.org/wiki/Kernel_(statistics)>.

[6] Konvoluce. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2011 [cit. 2012-04-15]. Dostupné z: <http://cs.wikipedia.org/wiki/Konvoluce>

[7] NVIDIA. *NVIDIA CUDA C Programming Guide 4.1* [online]. November 2011. Dostupné z WWW: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>

[8] SANDERS, Jason. *CUDA by example: an introduction to general-purpose GPU programming*. 1st print. Upper Saddle River: Addison-Wesley, 2010, 290 s. ISBN 978-0-13-138768-3.